Optimization of CUDA GPU Kernels and Translation to AMDGPU in Polygeist/MLIR

Ivan R. Ivanov[1,2]

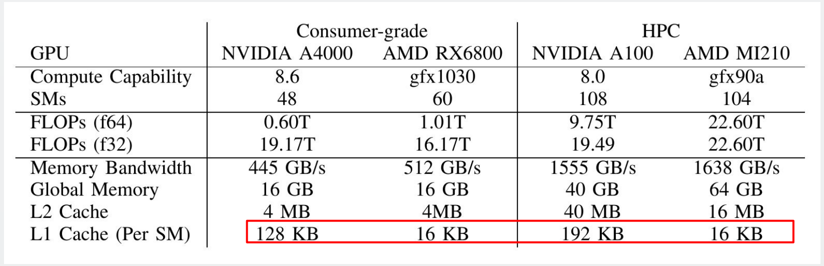
Alex Zinenko[3], Jens Domke[2], Endo Toshio[1], William S. Moses[4]

1. Tokyo Institute Of Technology
2. RIKEN R-CCS
3. Google
4. UIUC

Motivation

- Writing high performance CUDA code is **hard**
- Even more difficult to make it portable

  Big differences in GPU archs - warp size is 16, 32, 64 on Intel,NVIDIA, AMD



	Consumer-grade		HPC	
GPU	NVIDIA A4000	AMD RX6800	NVIDIA A100	AMD MI210
Compute Capability	8.6	gfx1030	8.0	gfx90a
SMs	48	60	108	104
FLOPs (f64)	0.60T	1.01T	9.75T	22.60T
FLOPs (f32)	19.17T	16.17T	19.49	22.60T
Memory Bandwidth	445 GB/s	512 GB/s	1555 GB/s	1638 GB/s
Global Memory	16 GB	16 GB	40 GB	64 GB
L2 Cache	4 MB	4MB	40 MB	16 MB
L1 Cache (Per SM)	128 KB	16 KB	192 KB	16 KB

- Large amount of (legacy) scientific C/C++ CUDA code
- **Large cost** of porting and tuning to another GPU architecture (or vendor)

Aim and Contributions

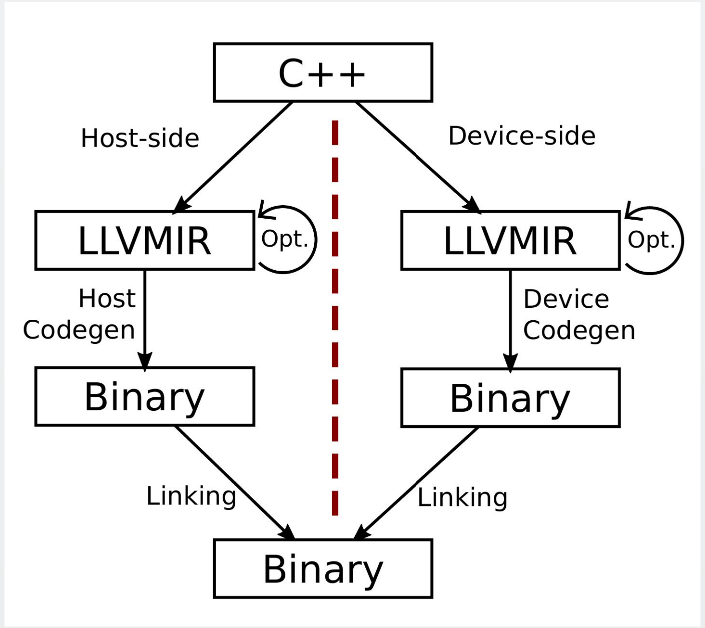
Write simple CUDA code once - let the compiler optimize and tune it for the target architecture (even AMD GPUs!)

How?

- **Represent parallel GPU computation in MLIR** (Extend Polygeist)
- Optimizations for **maximizing target hardware utilization**
- **CUDA to AMDGPU translation**

Traditional compilers

Device and Host side code split at the start of the pipeline (Example: clang)



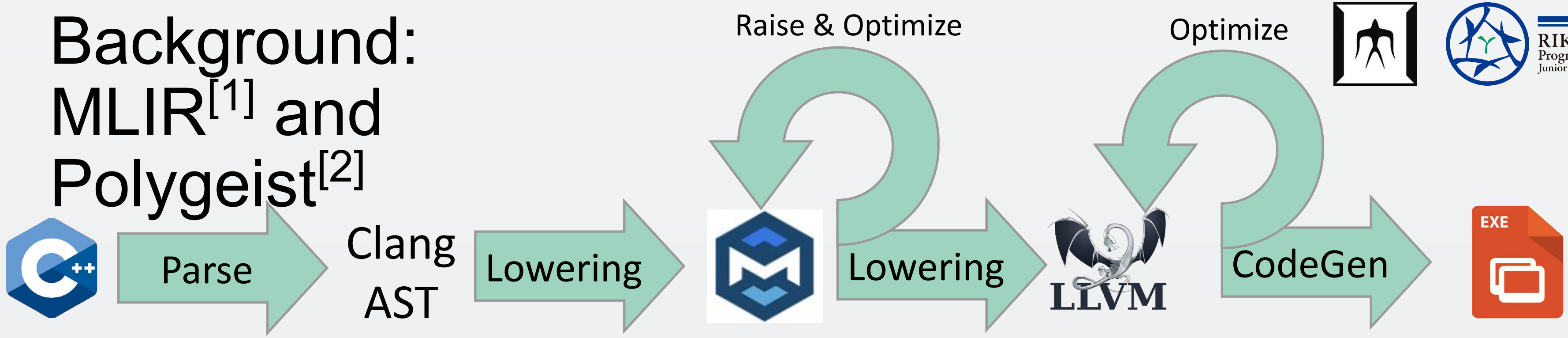
The information about the parallelism is hidden behind a library call

```
; Host side
declare @foo
define void @main() {
  ...
  call @cudaLaunchKernel(... , @foo)
}
```

```
; Device side
define void @foo(%a, %b, %c, %d, %i) {
  %s = call @sqrt(%d) ; Executed N times
  c[%i] = a[%i] + b[%i] + %s
}
```

The single thread work, where is the parallelism?

Background: MLIR[1] and Polygeist[2]



- Generic C and C++ frontend that generates "standard” and user-defined MLIR (templates, classes, unions, etc. all supported)
- Preserves the structure of programs (parallelism, control flow, etc)
- Collection of high-level optimization and analysis passes

[1] MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. CGO’21.
[2] Polygeist: Raising C to Polyhedral MLIR; Moses, Chelini, Zhao, and Zinenko. PACT ’21.

Optimization Friendly Parallel Representation in MLIR

GPU computation with **parallel semantics** and **context**

device region

grid

shared memory

block

synchronisation

```
func @launch(%h_out : memref<?xf32>,
             %h_in  : memref<?xf32>, %n : i64) {
  // Host code
  %d_out = gpu.alloc ...
  %d_in = gpu.alloc ...
  gpu.memcpy %d_in %h_in
  polygeist.gpu_region {
    // Device code
    parallel.for (%bx, %by, %bz) = (0, 0, 0)
                 to (grid.x, grid.y, grid.z) {
      %shared_val = memref.alloca : memref<f32>
      parallel.for (%tx, %ty, %tz) = (0, 0, 0)
                   to (blk.x, blk.y, blk.z) {
        if %tx == 0 {
          %sum = func.call @sum(%d_in, %n)
          memref.store %sum, %shared_val[] : memref<f32>
        }
        polygeist.barrier(%tx, %ty, %tz)
        %tid = %bx + blk.x * %tx
        if %tid < %n {
          %res = ...
          memref.store %res, %d_out[%tid] : memref<?xf32>
        }
      }
    }
  }
  gpu.memcpy %h_out %d_out
}
```

- Easier parallel optimizations
- Host-device cross-optimizations
- Tweaking kernel launch configuration

[1] High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs. William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. (PPoPP '23).

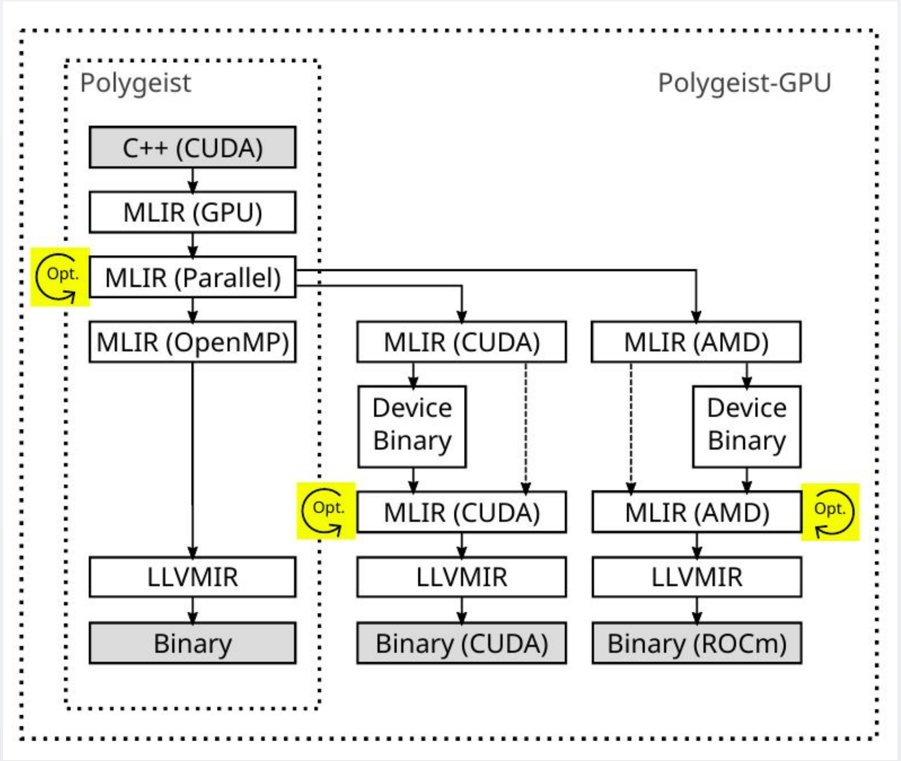
Support for GPU Compilation in Polygeist: The pipeline

Added

- Parallel -> Generic GPU

Adapted from upstream MLIR

- Generic GPU -> CUDA
- Generic GPU -> AMDGPU



Thread coarsening

Combining multiple threads worth of work in a single one (and interleaving)

```
parallel %i = from 0 to 16 {
  A(%i)
  B(%i)
  C(%i)
}
```

```
parallel %i = from 0 to 8 {
  %i_0 = %i * 2
  %i_1 = %i * 2 + 1
  A(%i_0)
  A(%i_1)
  B(%i_0)
  B(%i_1)
  C(%i_0)
  C(%i_1)
}
```

+ Better instruction level parallelism, result re-use, etc..

– Worse memory access patterns…

Recursive Unroll and Interleave

Generic recursive thread coarsening that works on parallel loops

```
parallel %i = from 0 to 16 {
  ...
  for %j = from 0 to 32 {
    A(%i, %j)
    B(%i, %j)
  }
}
```

```
parallel %i = from 0 to 8 {
  ...
  for %j = from 0 to 32 {
    A(%i_0, %j)
    A(%i_1, %j)
    B(%i_0, %j)
    B(%i_1, %j)
  }
}
```

Recursive Unroll Interleave

We have a generic version of thread coarsening…

We can apply it to blocks!

```
parallel %block = 0 to %n {
  %shmem = alloca()
  parallel %thread = 0 to 1024 {
    A(%block, %thread)
    B(%block, %thread)
  }
}
```

⇨

```
parallel %block = 0 to (%n / 2) {
  %block_0 = %block * 2
  %block_1 = %block * 2 + 1
  %shmem_0 = alloca()
  %shmem_1 = alloca()
  parallel %thread = 0 to 1024 {
    A(%block_0, %thread)
    A(%block_1, %thread)
    B(%block_0, %thread)
    B(%block_1, %thread)
  }
}
```

+ Nicer memory access patterns
+ **Finer grained** control over the **unroll factor** (no need to be a divisor of the bound)
  we instead generate and epilogue loop (a new kernel)

Recursive Unroll Interleave

We have a generic version of thread coarsening…

We can apply it to blocks!

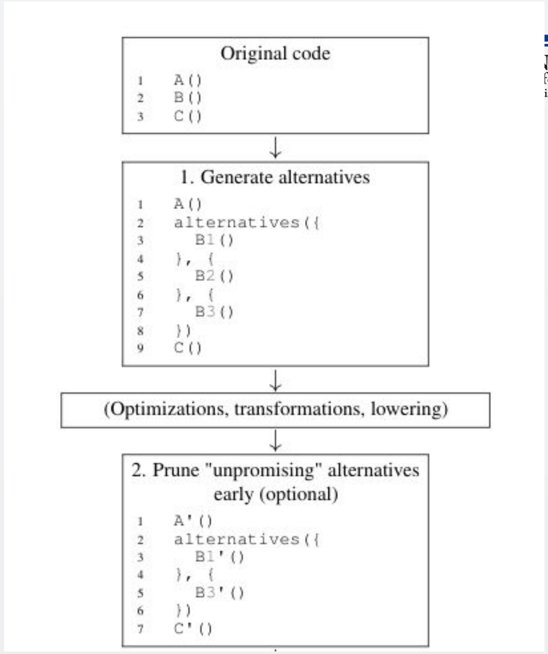
We can apply it to threads!

We can **combine** the two approaches

How do we choose the best configuration from all the possible combinations?

Alternative Code Paths

- **Multiple versions** of code that achieve the **same result** (different {block, thread} coarsening) in the parallel representation
- Defer choosing best one until later in the pipeline
  - Lower down to gpu binaries
  - Gather statistics about the kernel (registers used, memory spilled, theoretical occupancy, etc.)
  - Discard unpromising alternatives



Original code
1. Generate alternatives
(Optimizations, transformations, lowering)
2. Prune "unpromising" alternatives early (optional)

```
A()
B()
C()
```

```
A()
alternatives({
  B1()
}, {
  B2()
}, {
  B3()
})
C()
```

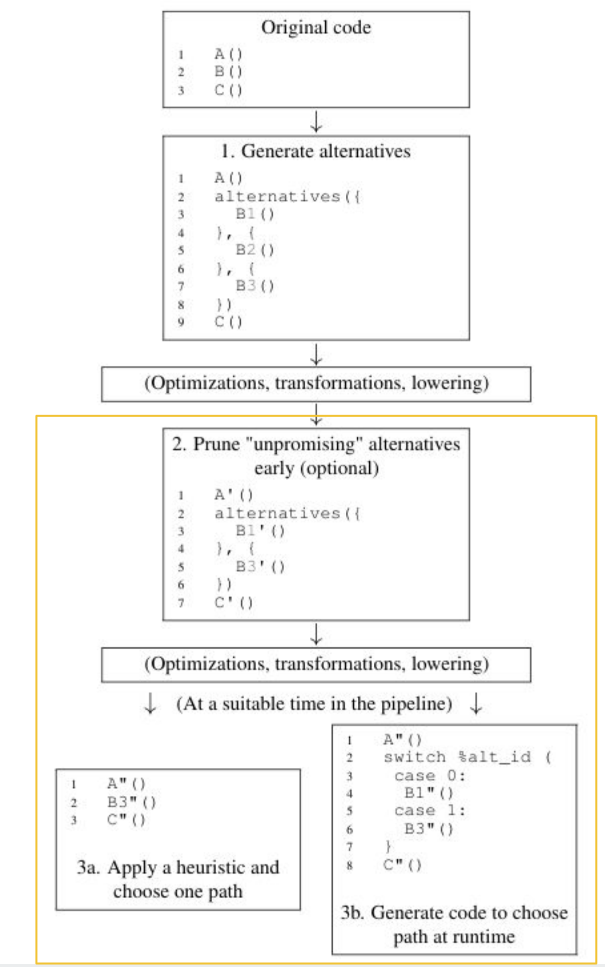
```
A'()
alternatives({
  B1'()
}, {
  B3'()
})
C'()
```

Alternative Code Paths

For the left over alternatives:

Easy to use Timing Driven Optimisation workflow

- Compile in *profiling* mode - the compiler generates N versions
- Run program N (or more) times to collect profiling data
- Compile in *optimization* mode - the compiler picks the best configuration for you


Original code
1. Generate alternatives
(Optimizations, transformations, lowering)
2. Prune "unpromising" alternatives early (optional)
(Optimizations, transformations, lowering)
(At a suitable time in the pipeline)
3a. Apply a heuristic and choose one path
3b. Generate code to choose path at runtime

```
A()
B()
C()
```

```
A()
alternatives({
  B1()
}, {
  B2()
}, {
  B3()
})
C()
```

```
A'()
alternatives({
  B1'()
}, {
  B3'()
})
C'()
```

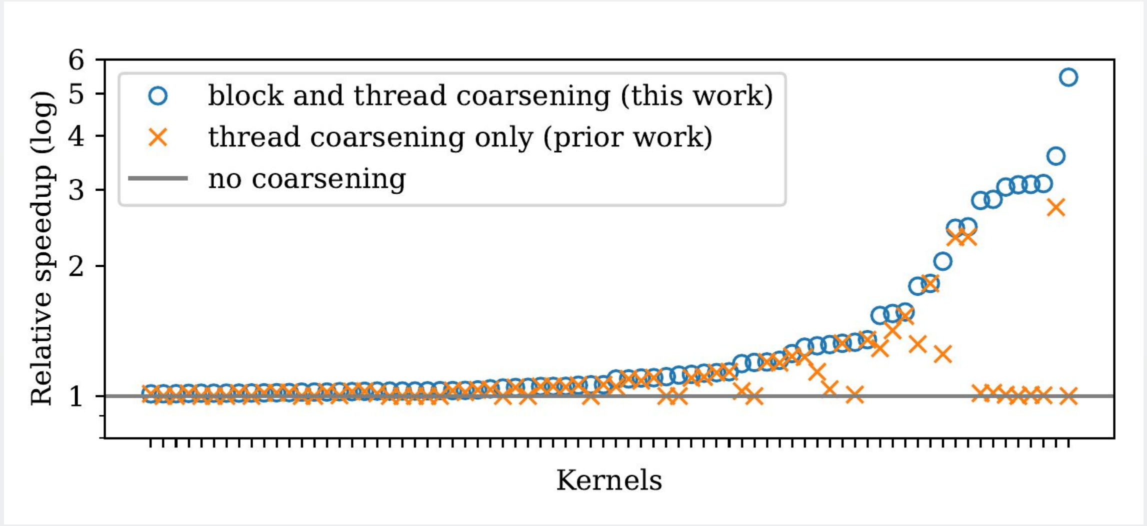
```
A"()
B3"()
C"()
```

```
A"()
switch %alt_id (
 case 0:
  B1"()
 case 1:
  B3"()
}
C"()
```

Evaluation

Evaluation

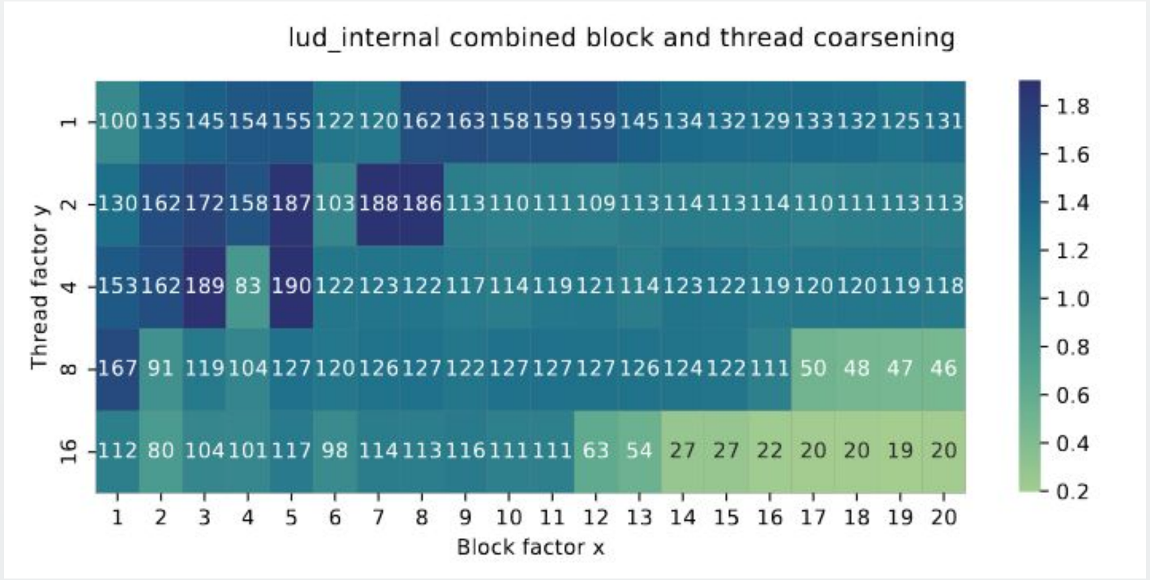
On the supported HeCBench and Rodinia CUDA kernels



(best configuration, only the ones with speedup > 1%)

Some interesting results…

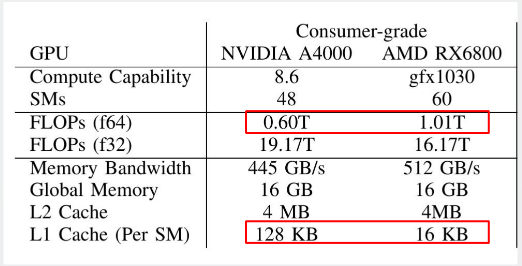
Finer granularity of block coarsening needed to maximise performance



Thread factor y \ Block factor x	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	100	135	145	154	155	122	120	162	163	158	159	159	145	134	132	129	133	132	125	131
2	130	162	172	158	187	103	188	186	113	110	111	109	113	114	113	114	110	111	113	113
4	153	162	189	83	190	122	123	122	117	114	119	121	114	123	122	119	120	120	119	118
8	167	91	119	104	127	120	126	127	122	127	127	127	126	124	122	111	50	48	47	46
16	112	80	104	101	117	98	114	113	116	111	111	63	54	27	27	22	20	20	19	20

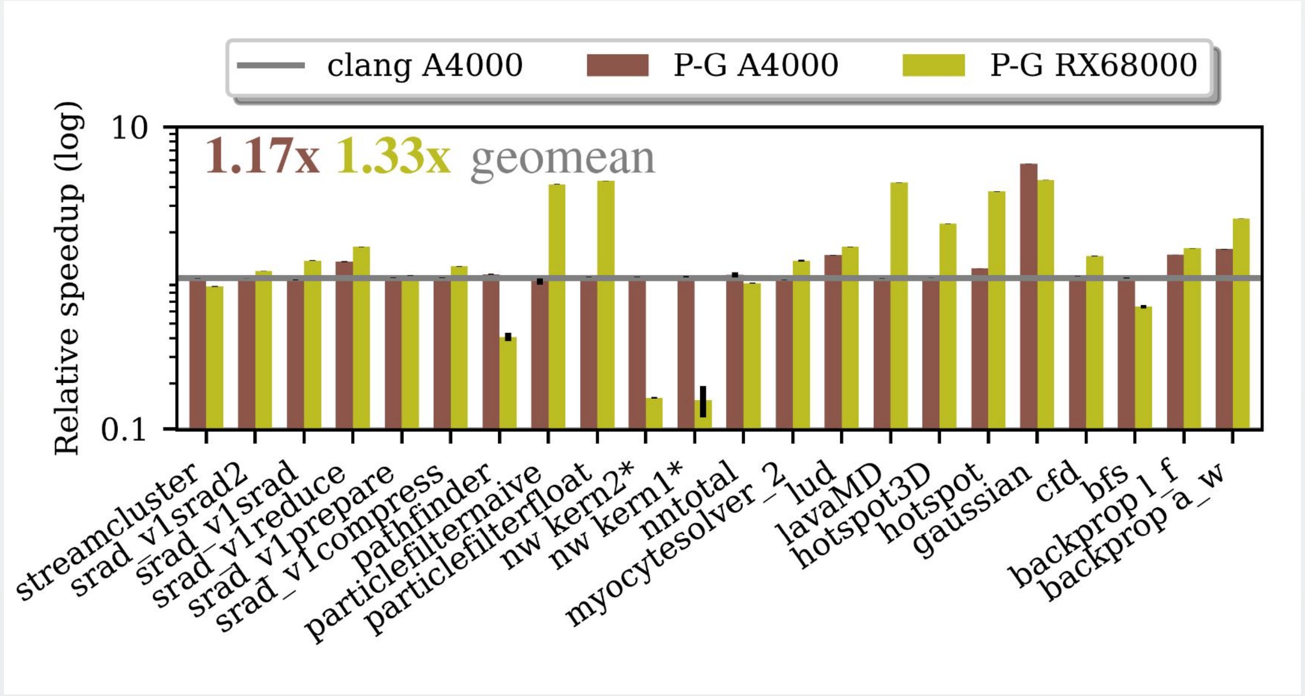
Unroll Factor of 5? 3?

CUDA vs AMD GPU



GPU	Consumer-grade NVIDIA A4000	AMD RX6800
Compute Capability	8.6	gfx1030
SMs	48	60
FLOPs (f64)	0.60T	1.01T
FLOPs (f32)	19.17T	16.17T
Memory Bandwidth	445 GB/s	512 GB/s
Global Memory	16 GB	16 GB
L2 Cache	4 MB	4MB
L1 Cache (Per SM)	128 KB	16 KB

Rodinia benchmarks



Conclusion

Performance portability of CUDA code

- Parallel transformations to best utilise available GPU resources
- Timing Driven Optimization framework
- Translation layer to AMD GPU

Nice representation of the computation

⇩

Better (and easier) parallel optimizations